Micro Digital

smxFLog™ User's Guide

Flash Logger

Version 1.00
December 12, 2007

by Yingbo Hu and David Moore

Table of Contents

1. Overview 1

1.1 Restrictions 1

2. Using smxFLog 2
2.1 Installation 2
2.2 Getting Started 2
2.3 Basic Terms 2
2.4 Configuration Settings 2
2.5 Record Pointers and Marks 3
2.6 Partitioning the Flash 3
2.7 Mixed and Small Block Sizes 5
2.8 Wear Leveling 5
2.9 Power Fail Safety 5

3. Flash Logger API 6
3.1 API Data Types 6
3.2 API Reference 6

4. Flash Hardware IO Routines 11

5. Application Notes 12
5.1 Offload Log Data to smxFS 12
5.2 Offload Log Data to smxUSBD Serial Device 12
5.3 Erase Oldest Record When System is Idle 13
5.4 NAND Flash Array 13
5.5 Multiple Logs and Record Types 13

A. File Summary 15

B. Porting Notes 16
B.1 C Library Function Requirements 16
B.2 OS System Call Requirements 16

C. Size and Performance 17
C.1 Code Size 17
C.2 Data Size 17

D. Tested Hardware 18
D.1 NAND 18
D.2 NOR 18

© Copyright 2007

Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

All rights reserved.

smxFLog is a Trademark of Micro Digital Inc.
smx is a Registered Trademark of Micro Digital Inc.

1. Overview

smxFLog has the simple purpose of logging data efficiently and reliably in flash, using a minimum of RAM. Logging data is a very common operation in embedded systems, and warrants a good solution. A full filesystem is not well-suited to this purpose. It makes sense to use smxFLog in addition to a full filesystem, because both have advantages.

Logging data is a sequential operation of appending data to a file. This is not efficient in a DOS/Windows FAT filesystem, particularly when writing to flash media. The problem is that writing less than a full cluster of data each time requires the partially-filled last cluster to be moved around in flash each time, and also requires modifying FAT and directory sectors, which also have to be moved. This hurts peformance and wears the flash. It also means garbage collection is frequently required, which is a lengthy operation, which can stall further logging temporarily. smxFLog can append new records without moving data, and has no tables to move either.

smxFLog is also power-fail-safe because of its simplicity. The DOS/Windows FAT filesystem is inherently not power-fail-safe, unless it is extended with journaling or some other mechanism, which might be incompatible. Non-standard filesystems can be designed to be power-fail-safe, as smxFFS was, but it requires a lot of RAM to make it so. This makes it unsuitable for many low-end SoCs with only on-chip SRAM and no ability to connect to external memory.

Filesystems are very useful, though, because they allow storing multiple files and directories. Moreover, the DOS/Windows FAT filesystem allows media to be shared with other computers. Even non-removable media in an embedded target can be read from the filesystem on a PC if the target is running a USB device stack with mass storage driver, such as smxUSBD. In this case, a disk stored in resident flash will look like a disk to a PC connected by USB cable. Thus, filesystems are useful because they allow sharing data either using removable media or via a data link such as USB or FTP.

It is beneficial to use smxFLog and smxFS in the same system. Data can be logged reliably by smxFLog and periodically offloaded in chunks to the filesystem, which is efficient. API functions are provided, and code examples shown in section 5 to make this easy. They can coexist in the same flash (in separate partitions), and in this case, they will share the same low-level flash driver, so there is only one driver to port and no duplicated code.

1.1 Restrictions

In order to make smxFLog simple, efficient, and reliable, the following restrictions are imposed:

- Each data record must have the same size, specified at compile time.
- New data records can only be appended following the old data records, and existing data records cannot be modified.
- Supported NAND flash types are those that: support block erase (changing all cells to 0xFF), support partial page write at least 3 times, and have a spare area. There is no overhead in the data area.
- Supports all NOR flash types. There is overhead in the data area to store the status and ECC bytes since there are no spare areas.
- No support for multiple logs or record types. This can be accomplished by the application. See section 5.5. Multiple Logs and Record Types.

2. Using smxFLog

2.1 Installation

smxFLog is installed by copying files from the distribution media. When ordered with the SMX® RTOS, it is part of the SMX release and is installed with it.

2.2 Getting Started

smxFLog is configured to support any environment. If you are using a new compiler which is not in our porting file, see Appendix B Porting Notes, and implement the porting layer for your environment first, before using smxFLog.

2.3 Basic Terms

Block Minimum erasable unit of the flash chip. Some NOR flash chips may use the term *sector*, instead.
Page Maximum read/write unit of data of a NAND flash chip. It is 512 or 2048 bytes.
Record The log data unit size. All must be the same and power of 2.

2.4 Configuration Settings

If you change any settings you should rebuild the smxFLog library, clean.

2.4.1 flcfg.h

flcfg.h contains flash logger configuration constants that allow you select features and tune performance, code size, and RAM usage.

SFL_NAND
Set to “1” to use NAND flash to log data.

SFL_NOR
Set to “1” to use NOR flash to log data.

SFL_MULTITASKING
Set to “1” to enable multitasking reentry protection. Default settings is “0”

SFL_USE_ECC
Set to “1” to enable ECC code to check and correct data consistency. Default setting is “0”.

SFL_READBACK_VERIFY
Set to “1” to enable read back verification to check data consistency. It will allocate an additional record-sized read back buffer in RAM. Default setting is “0”.

SFL_RECLAIM_RECORD
Set it to “1” to enable auto reclaim of the oldest record when the flash is full. Default setting is “1”. If 0, smxFLog will stop logging when the flash is full.

SFL_RECORD_SIZE

This specifics the record size. Must be power of 2. Default setting is 512 for NAND flash and 64 for NOR flash.

Data records must all be the same size, and padding must be added if the actual data size is less than the record size. For small data records, we recommend writing multiple data records to each flash record.

NAND flash: Data records must be 512 bytes or a multiple of 512 bytes. This is because of partial programming limitations. If the record were 256 bytes, there would be 2 per page, requiring partial programming 6 times (3 times for each status byte), but many flash chips support only 3 times. If the flash chip supports at least 6 times, this can be reduced to 256.

NOR flash: Data records can be any size because there is no partial programming issue, but the overhead of status and ECC bytes becomes significant as block size decreases. It is important to support smaller record sizes for NOR flash since most systems won't have much. The minimum tested record size is 32 bytes.

2.4.2 flport.h

flport.h contains compiler and OS definitions. See Appendix B Porting Notes about these.

The following basic types are used in the code:

u8	unsigned 8-bit
u16	unsigned 16-bit
u32	unsigned 32-bit
uint	unsigned int

2.5 Record Pointers and Marks

This manual does not document the theory of operation of smxFLog. However, we explain here a few key concepts that will aid in understanding how to use the API.

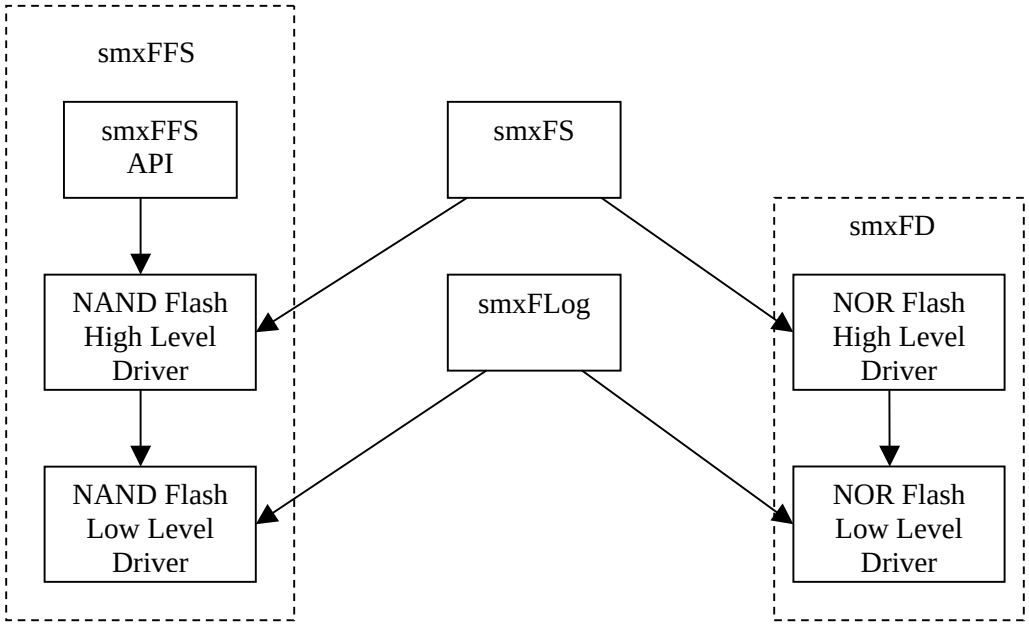
smxFLog maintains pointers to the oldest record, next record to read, and the next record to write. These are initialized by scanning the whole flash at startup. The oldest record pointer is advanced as old flash blocks are reclaimed. The read pointer is advanced after each sfl_Read() to point at the next record to read. The write pointer is advanced after each sfl_Write() to point at the next empty slot to write to. The flash is treated as a circular queue and the pointers go round and round.

sfl_ReadPtrMark() allows marking the record the read pointer currently points to so that if a power fail occurs, the read pointer will start there after restart.

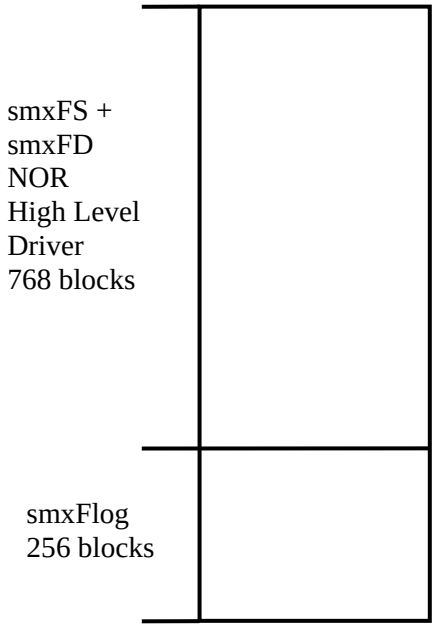
2.6 Partitioning the Flash

By default smxFLog uses the entire flash. However, it is easy to reserve blocks before it and after it in the flash, for use by other software, even our FAT filesystem smxFS, as is discussed in the Overview of this manual. In order to understand how to configure this, the following diagrams are helpful.

The following diagram shows how the different SMX file systems relate to each other.



Notice that smxFLog is at the same level as the NAND and NOR high level drivers. The start and end block numbers in flashcnf.h (NAND) and fdcfg.h (NOR) are comparable to those in flcfg.h (smxFLog). The following diagram shows how smxFS + smxFD and smxFLog can share NOR flash. This example shows a flash chip with 1024 flash blocks.



To achieve this, configure as follows:

fdcfg.h (smxFD; NOR driver for smxFS)
```
#define NOR_START_BLOCK_NUM     0
#define NOR_END_BLOCK_NUM       256
```

flcfg.h (smxFLog)
```
#define SFL_START_BLOCK_NUM     768
#define SFL_END_BLOCK_NUM       0
```

Note that additional blocks could be reserved at the start and end of flash for other purposes by changing NOR_START_BLOCK_NUM and SFL_END_BLOCK_NUM to non-zero values.

Partitioning the NAND flash is similar.

2.7 Mixed and Small Block Sizes

For NAND flash, all blocks are the same size, but it is common for NOR flash to have smaller block sizes at the start or end of the flash (i.e. the boot block). The total size of these is the same as a normal flash block (e.g. 64KB). To handle this, you can either exclude these small blocks from the partitions of the flash used by smxFD and smxFLog or you can treat them as a single block by adding special handling to nor_IO_SectorRead() and nor_IO_SectorWrite() in the low-level NOR driver.

SoCs with on-chip flash typically have small blocks, such as 4KB instead of the typical 64KB. The NOR driver can support the actual block size or combine them into larger blocks, if desired. [TODO: advantage?]

2.8 Wear Leveling

Wear leveling is guaranteed because data is written to the flash sequentially.

2.9 Power Fail Safety

Power fail safety is easily achieved because there are no data structures, such as mapping tables, to keep consistent with the data. A status byte for each record indicates whether a write operation is pending or completed. Bad records (e.g. partially written or bad ECC) are simply skipped.

3. Flash Logger API

The smxFLog API is defined in smxflog.h, which contains the functions that are called by the application.

int **sfl_Init**(uint iChipID, uint iFlag);

int **sfl_Release**(uint iChipID);

int **sfl_Read**(uint iChipID, u8 *pRecord, uint iNumRecords);

int **sfl_ReadPtrMark**(uint iChipID, uint iEraseOldRecords);

int **sfl_ReadPtrRestore**(uint iChipID);

int **sfl_Write**(uint iChipID, u8 *pRecord);

int **sfl_Erase**(uint iChipID, uint Flag);

3.1 API Data Types

These are defined in **flport.h**. See 2.4.2 flport.h.

3.2 API Reference

int **sfl_Init** (uint iChipID, uint iFlag)

Summary Initialize smxFLog hardware and internal data structures.

Descr This function should be called first before you use smxFLog. It calls the NAND or NOR Flash Hardware IO Routine to initialize the flash chip and retrieve the basic information of the Flash Chip such as the block size and total number of blocks. It then tries to scan the flash chip to find the record pointers.

Pars

nChipID — The device ID you want to use. Only pass 0 for the current version.

iFlag — 0 or one of the following flags which will be used to speed up the initialization procedure.

0
Do normal power up check when you still need the old records on the flash. It may delay your power up procedure because it will scan the whole flash area to initialize the record pointers.

SFL_INIT_ERASE_ALL
Erase the whole flash area before using it. If you pass this flag, all the old records will be lost.

SFL_INIT_SKIP_CHECK
Do not scan the flash to initialize the record pointers. Only pass this flag when your application has already erased the whole flash area. It is the fastest way to initialize smxFLog.

Returns

0 Initialization succeeded.
<0 Initialization failed.

See Also sfl_FRelease()

Example

```
if(0 == sfl_Init(0, 0))
    printf("Flash Logger Initialized.");
```

int **sfl_Release** (uint iChipID)

Summary Release smxFLog resources.

Descr This function should be called when you are done with smxFLog. It resets the internal record pointer set by sfl_Init() and calls the NAND or NOR Flash Hardware IO Routine to release the hardware resources.

Pars nChipID The device ID you want to use (the same ID you passed to sfl_Init()).

Returns 0 Release succeeded.
<0 Release failed.

See Also sfl_Init()

Example

```
if(0 == sfl_Release(0))
    printf("Flash Logger Released.");
```

int **sfl_Read**(uint iChipID, u8 *pRecord, uint iNumRecords)

Summary Read one or more records from the flash memory.

Descr This function will read some records from the flash memory. The buffer should be allocated by the application, so make sure the buffer is big enough to hold the specified number of records. The read pointer (in RAM) is advanced to point to the next record to read.

Pars

nChipID	The device ID you want to use (the same ID you passed to sfl_Init()).
pRecord	The buffer for the records.
iNumRecords	The number of records.

Returns The actual number of records the flash logger read from the flash. If there are no unread records, this API will return 0.

See Also sfl_Init(), sfl_Write()

Example

```
u8 RecordBuf[SFL_RECORD_SIZE];
if(sfl_Read(0, RecordBuf, 1))
    printf("Got one record.");
else
    printf("No more records left");
```

int **sfl_ReadPtrMark**(uint iChipID, uint iEraseOldRecords)

Summary Mark the read pointer in the flash chip.

Descr This function marks flash record to which the current read pointer (stored in RAM) points, so the next time you power on smxFLog, you don't need to retrieve the old records you already read from the flash. If iEraseOldRecords is true then the old data records before it will be erased. But it will only erase up to the nearest physical block boundary below the new read pointer. This is useful to pre-clear the oldest block(s) of records during idle time

Pars

nChipID	The device ID you want to use (the same ID you passed to sfl_Init()).
iEraseOldRecords	Specifies whether to erase the old records after the read pointer has been marked.

Returns

0 Mark succeeded.
<0 Mark failed.

See Also sfl_Read()

Example

```
u8 RecordBuf[SFL_RECORD_SIZE];
if(sfl_Read(0, RecordBuf, 1))
    sfl_ReadPtrMark(0, 1);
else
    printf("No more records left");
```

int **nor_ ReadPtrRestore** (uint iCheckID)

Summary Restore the read pointer to the last marked position.

Descr This function restores the read pointer to the last position stored in the flash so if anything goes wrong, the application can restart reading from the last known read pointer.

Pars nChipID The device ID you want to use (the same ID you passed to sfl_Init()).

Returns

0 Restore succeeded.
<0 Restore failed.

See Also sfl_Read(), sfl_ReadPtrMark()

Example

```
u8 RecordBuf[SFL_RECORD_SIZE];
sfl_ReadPtrMark(0, 1);
if(sfl_Read(0, RecordBuf, 1))
    sfl_ReadPtrRestore(0);
else
    printf("No more record left");
```

int **sfl_Write**(uint iChipID, u8 *pRecord)

Summary Append a new record to the flash.

Descr This function adds a new record to the flash following the previous one. The write pointer is advanced to the next empty record.

Pars

nChipID	The device ID you want to use (the same ID you passed to sfl_Init()).
pRecord	The pointer to the record data to write.

Returns

1	Record data appended.
0	Flash is full and you did not enable automatic reclaim of the oldest record.

See Also sfl_Read(), sfl_Erase()

Example

```
u8 RecordBuf[SFL_RECORD_SIZE];
sfl_Write(0, RecordBuf);
```

int **sfl_Erase**(uint iChipID, uint Flag)

Summary Erase one or more of the oldest records

Descr This function erases one or more oldest records.

Pars

nChipID	The device ID you want to use (the same ID you passed to sfl_Init()).
Flag	One of the following flags to indicate how many of the oldest records should be erased: **SFL_ERASE_ONE_BLOCK** Erase only the oldest block of records each time. It is good to call it in the idle task/loop if you want to keep logging and don't want to decrease performance **SFL_ERASE_ALL_OLD_BLOCKS** Erase all the old records up to the one the current pointer is in. At least one block will be kept. **SFL_ERASE_ALL_BLOCKS**

Erase all data records. After this call the flash chip is just like a new chip; all the blocks will be erased. smxFLog will write new records starting at the beginning of the flash (or partition).

Returns Number of deleted records.

See Also sfl_Init()

Example

```
sfl_Erase(0, SFL_ERASE_ONE_BLOCK);
```

4. Flash Hardware IO Routines

smxFLog uses the same low-level drivers as smxFFS and smxFD. Please check the smxFFS User's Guide and smxFD User's Guide for the details about those routines.

5. Application Notes

5.1 Offload Log Data to smxFS

The data records can be written to a file in a file system by smxFS, so the user can offload the log data to removable media such as a thumb drive, or it can be retrieved from the file system later via data link such as USB, FTP, etc.

```
#include "smxflog.h"
#include "smxfs.h"

void SendRecordToFile(void)
{
    FILEHANDLE fp;
    int iRecordNum;
    char szRecord[4*SFL_RECORD_SIZE];

    fp = sfs_fopen("A:\\DataRecord.bin", "wb");
    if(fp)
    {
        do
        {
            iRecordNum = sfl_Read(0, szRecord, 4); /* read 4 records each time */
            sfs_fwrite (szRecord, iRecordNum*SFL_RECORD_SIZE, 1, fp);
        } while(iRecordNum > 0);
        sfs_fclose(fp);

        /* Done offloading, so set the read pointer in flash to the current record and erase all the records
           before it (up to the current block). */
        sfl_ReadPtrMark(0, 1);
    }
}
```

5.2 Offload Log Data to smxUSBD Serial Device

Data records can be retrieved through the smxUSBD serial port emulator, so the user can use a laptop to get the log data.

```
#include "smxflog.h"
#include "smxusbd.h"

void SendRecordToUSBSerial(void)
{
    char szRecord[SFL_RECORD_SIZE];
    while(sfl_Read(0, szRecord, 1) == 1)
    {
        sud_SerialWriteData(0, szRecord, SFL_RECORD_SIZE);        /* 512 byte data record */
    }
}
```

5.3 Erase Oldest Record When System is Idle

The application should erase old blocks when idle so that logging is never delayed. Only erase 1 block at a time if the application needs to continue logging.

```
void idle_task_main(void)
{
    /* do other idle job here */
    …
    /* now try to reclaim the oldest block of records */
    sfl_Erase(SFL_ERASE_ONE_BLOCK);
}
```

5.4 NAND Flash Array

To get a bigger capacity, you can create a NAND flash array. For details please refer to the smxFFS User's Guide, Appendix B: Flash Chip Array.

5.5 Multiple Logs and Record Types

smxFLog itself does not support multiple logs or record types. There is a single log containing only one type of record. This makes smxFLog simple and reliable. To achieve something close to multiple logs, store a Log ID in each record. Similarly, if you wish to have different record types, store a Record Type value in each record.

```
#include "smxflog.h"
#include "smxfs.h"

typedef struct
{
    uint data1;
    uint data2;
    uint data3;
    u8   data4[12];
} LOG_TYPE1;

typedef struct
{
    u8   data1[8];
    uint data2;
    uint data3;
    u8   data4[4];
} LOG_TYPE2;

#define TYPE1_ID  0x01
#define TYPE2_ID  0x02

int WriteLog1(LOG_TYPE1 *pLog1)
{
    u8 Record[SFL_RECORD_SIZE];
    memset(Record, 0, SFL_RECORD_SIZE);
    Record[0] = TYPE1_ID; /* first 4 bytes of each record is the log ID */
    memcpy(&Record[4], pLog1, sizeof(LOG_TYPE1));
    return sfl_Write(0, Record);
}
```

```
int WriteLog2(LOG_TYPE2 *pLog1)
{
    u8 Record[SFL_RECORD_SIZE];
    memset(Record, 0, SFL_RECORD_SIZE);
    Record[0] = TYPE2_ID; /* first 4 bytes of each record is the log ID */
    memcpy(&Record[4], pLog1, sizeof(LOG_TYPE2));
    return sfl_Write(0, Record);
}

void SendRecordToFile(void)
{
    FILEHANDLE fp1;
    FILEHANDLE fp2;
    int iRecordNum;
    char szRecord[SFL_RECORD_SIZE];
    fp1 = sfs_fopen("A:\\Record1.bin", "ab");
    fp2 = sfs_fopen("A:\\Record2.bin", "ab");
    if(fp1 && fp2)
    {
        do
        {
            iRecordNum = sfl_Read(0, szRecord, 1);
            if(iRecordNum)
            {
                if(szRecord[0] == TYPE1_ID)
                    sfs_fwrite (szRecord, SFL_RECORD_SIZE, 1, fp1);
                else if(szRecord[0] == TYPE2_ID)
                    sfs_fwrite (szRecord, SFL_RECORD_SIZE, 1, fp2);
            }
        } while(iRecordNum > 0);
        sfs_fclose(fp1);
        sfs_fclose(fp2);
        /* Done offloading, so set the read pointer in flash to the current record
           and erase all the records before it (up to the current block). */
        sfl_ReadPtrMark(0, 1);
    }
}
```

A. File Summary

FILE	DESCRIPTION
flcfg.h	Configuration file for smxFLog.
flport.h flport.c	Porting definitions, macros, and functions for compiler and OS. Ported to SMX, as shipped.
flog.c	Flash Logger API Implementation.
smxflog.h	Flash Logger API.
flhdw.h	NAND Flash Hardware IO routines.
norio.h	NOR Flash Hardware IO routines.

B. Porting Notes

The porting layer is simple. Only mutex APIs need to be implemented if you set SFL_MULTITASKING to “1”.

B.1 C Library Function Requirements

smxFLog uses the following C library functions. You must implement them if your compiler does not provide.

- memcmp()
- memcpy()
- memset()

B.2 OS System Call Requirements

If you want to run smxFLog in a multitasking environment, you need to implement some mutex APIs.

Those APIs are:

```
SFL_MUTEX_HANDLE SFL_MUTEX_CREATE (void);
void        SFL_MUTEX_RELEASE (SFL_MUTEX_HANDLE *handle);
int         SFL_API_ENTER (SFL_MUTEX_HANDLE *handle);
int         SFL_API_EXIT (SFL_MUTEX_HANDLE *handle);
```

C. Size and Performance

C.1 Code Size

Code size varies depending upon CPU, compiler, and optimization level.

	ARM7/9	**ColdFire**
	IAR	CodeWarrior
smxFLog (without ECC)	2 KB	2.5 KB
smxFLog (with ECC)	4 KB	5 KB

C.2 Data Size

smxFLog was designed to minimize RAM use.

smxFLog core	32 B
smxFLog ECC code (disabled by default)	256 B
smxFLog readback verify code (disabled by default)	record size

D. Tested Hardware

D.1 NAND

- K9F1G08U on NXP LPC2468 board. Record size: 512.
- K9F2808U on our Avnet Coldfire 5282 add-on board. Record size: 512.

D.2 NOR

- 39VF320 on NXP LPC2468 board. Record sizes: 32, 64, 128, 256, 512.
- 28F128K3, 28F256K3, 28F128J3D on MCF5485EVB board. Record sizes: 32, 64, 128, 256, 512, 1024.